UCRL-MA-150596

Unstructured Mesh Connectivity in UnstructuredMapping

K. K. Chand

October 22, 2002

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

Unstructured Mesh Connectivity in `UnstructuredMapping`

Kyle K. Chand Centre for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, 94551
chand1@llnl.gov
http://www.llnl.gov/casc/people/chand
http://www.llnl.gov/casc/Overture October 29, 2002

Abstract

The connectivity interface for `UnstructuredMapping` has been rewritten to provide a more thorough interface to the mesh. This new design also resembles the TSTT mesh query interface. While data is still stored in array form, indexed by integers, the interface provides iterators through the mesh entities and adjacencies. This document describes the additions to the `UnstructuredMapping` class as well as the definition and use of the `UnstructuredMappingIterator` and `UnstructuredMappingAdjacencyIterator` classes.

Contents

1 Introduction

Connectivity in the `UnstructuredMapping` is based on the set of "entities" specified in `UnstructuredMapping ::EntityTypeEnum`. "Vertex", "Edge", "Face", and "Region" refer to entities in ascending topological dimension, with "Vertex" equivalent to 0 and "Region" equivalent to 3:

```
enum EntityTypeEnum
{
  Invalid=-1,
  Vertex=0,
  Edge,
  Face,
  Region,
  Mesh, // kkc put this here to enable "Mesh" tagging...
  NumberOfEntityTypes
};
```

This approach will be familiar to users of the TSTT mesh query interface. Entities themselves are defined by their topological dimension (Type) and the specific vertices that they contain. For example, no two edges have the same vertices. Each entity has an orientation defined by their vertices and "Element" type as specified in `UnstructuredMapping ::ElementTypeEnum`:

```
enum ElementType
 {
  triangle,
  quadrilateral,
  tetrahedron,
  pyramid,
  triPrism,
  septahedron,
  hexahedron,
  other,
  boundary,
  NumberOfElementTypes
 };
```

ElementTypeEnum is analogous to the EntityTopology enum from the TSTT interface and specifies the "shape" of a given entity (triangle, quad, etc...). Iterating through the entities and their adjacencies is managed most effectively (though not exclusively) through the `UnstructuredMappingIterator` and `UnstructuredMappingAdjacencyIterator` classes. The Iterator classes provide iterations through entities of a specific type (eg all the Regions) while the AdjacencyIterators provide access to the entities adjacent to a specific entities. A tagging mechanism has also been added, again following the TSTT approach, in order to allow a very general method for placing information on entities and grouping them together into sets. Several additions to the methods and data in the `UnstructuredMapping` class have been created to facilitate the construction and use of the new connectivity and iterators.

Prior to discussing the details of the interface and changes to the `UnstructuredMapping` class, we demonstrate the basic use of the interface with two examples. In the first example we read an `UnstructuredMapping` from a file and then iterate through all the "Faces" in the mesh:

```
  UnstructuredMapping umap;
  umap.get(fileName);

  // loop through all the Faces in the mesh
  UnstructuredMappingIterator face;
  for ( face = umap.begin(UnstructuredMapping::Face);
        face!=umap.end(UnstructuredMapping::Face);
        face++ )
    cout<<"Here is a Face with index "<< *face <<endl;
```

The declaration and `get` method should be familiar to users of the `UnstructuredMapping` . Looping through all the faces in the `UnstructuredMapping` , however, is performed using the new `UnstructuredMappingIterator` iterator class. `UnstructuredMappingIterator` 's use mirrors that of STL containers and iterators. `UnstructuredMapping` 's `begin` and `end` methods require an argument specifying the type of entity requested (from `EntityTypeEnum`). The dereference operator of the iterator returns an integer index which may be used as part of an identification mechanism or access into another array. In order to obtain adjacency information (such as the vertices attached to the a specific face) we need to use an `UnstructuredMappingAdjacencyIterator` :

```
UnstructuredMappingIterator face;
for ( face =  umap.begin(UnstructuredMapping::Face);
      face != umap.end(UnstructuredMapping::Face);
      face++ )
  {
    cout<<"Face "<<*face<<" has vertices : ";

    UnstructuredMappingAdjacencyIterator faceVert;
    for ( faceVert =  umap.adjacency_begin(face, UnstructuredMapping::Vertex);
          faceVert != umap.adjacency_end(face, UnstructuredMapping::Vertex);
          faceVert++ )
       cout<<*faceVert<<"  ";

    cout<<endl;
  }
```

The `UnstructuredMapping` 's `adjacency_begin` and `adjacency_end` methods are used in the same fashion as the `UnstructuredMappingIterator` versions. They require an `UnstructuredMappingIterator` as the first argument in order to specify the "from" entity and a second argument to specify the type of the "to", or adjacent, entities. Variations on these two iterations will form the bulk of the examples and are expected to provide the necessary functionality to build unstructured mesh operators and manipulations.

2 Additions to the `UnstructuredMapping` class

2.1 Generic size and data access methods

- `const intArray & getEntities(EntityTypeEnum);`

  return an array of the vertices in all the entities of the type requested by the argument

- `inline int size( EntityTypeEnum t ) const;`

  returns the number of entities of type `t` available in the mesh.

- `inline int capacity( EntityTypeEnum t ) const;`

  return the total storage available for entities of type `t`.

- `int reserve( EntityTypeEnum, int) const;`

  reserve a specified amount of space (increase/decrease the capacity) for entities of a specified type

2.2 Iterator related methods

-
  ```
  inline UnstructuredMappingIterator
  begin(EntityTypeEnum entityType_, bool includeGhostEntities) const;
  ```

  returns an iterator pointing to the beginning of the list of entities of type entityType_. `includeGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included. NOTE: if the specified entity does not exist the mapping will attempt to build it!

-
  ```
  inline UnstructuredMappingIterator
  end(EntityTypeEnum entityType_, bool includeGhostEntities) const;
  ```

  returns an iterator pointing to the end of the list of entities of type`entityType_` . `includeGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included. NOTE: if the requested entity does not exist the mapping will try to build it!

-
  ```
  inline UnstructuredMappingAdjacencyIterator
  adjacency_begin(UnstructuredMappingIterator from,
                  EntityTypeEnum to, bool skipGhostEntities) const;
  ```

  returns an iterator pointing to the beginning of the list of entities of type `to` surrounding the entities specified by the iterator `from`. `skipGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included. NOTE: if the requested adjacency information does not exist then the mapping will attempt to build it.

-
  ```
  inline UnstructuredMappingAdjacencyIterator
  adjacency_end(UnstructuredMappingIterator from,
                EntityTypeEnum to, bool skipGhostEntities) const;
  ```

  returns an iterator pointing to the end of the list of entities of type `to` surrounding the entities specified by the iterator `from`. `skipGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included. NOTE: if the requested adjacency information does not exist then the mapping will attempt to build it.

-
  ```
  inline UnstructuredMappingAdjacencyIterator
  adjacency_begin(UnstructuredMappingAdjacencyIterator from,
                  EntityTypeEnum to, bool skipGhostEntities) const;
  ```
  returns an iterator pointing to the beginning of the list of entities of type `to` surrounding the entities specified by the iterator `from`. `skipGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included.

-
  ```
  inline UnstructuredMappingAdjacencyIterator
  adjacency_end(UnstructuredMappingAdjacencyIterator from,
                EntityTypeEnum to, bool skipGhostEntities) const;
  ```
  returns an iterator pointing to the end of the list of entities of type `to` surrounding the entities specified by the iterator `from`. `skipGhostEntities` is an optional argument; if true, the ghost entities in the iteration are skipped, if false (default) they are included.

2.3 Entity and connectivity manipulations

- `inline int maxVerticesInEntity(EntityTypeEnum type);`

  returns the maximum number of vertices in a given entity type.

- `inline ElementType computeElementType(EntityTypeEnum type, int e);`

  computes the element type as given in the `ElementType` enum for a `type` entity with index `e`.

- `inline int numberOfVertices(EntityTypeEnum, int);`

-
  ```
  bool
  entitiesAreEquivalent(EntityTypeEnum type, int entity, ArraySimple<int> &verticies);
  ```
  two entities are equivalent if their vertices are the same. This method returns true if the `type` entity with index `e` is equivalent to an entity specified by `vertices`.

- `void setAsGhost(EntityTypeEnum type, int entity);`

  toggles the `type` entity at index `entity` to be a ghost. This also adds a tag marking the entity as a ghost through the tagging mechanism. The entity is added to the list of entities with the same ghost tag.

- `bool specifyVertices(const realArray & verts);`

  provide a list of vertices to be copied into the mapping. This replaces the "node" array from the old interface. returns true if successful.

- `bool`
  `buildEntity(EntityTypeEnum type, bool rebuild, bool keepDownward, bool keepUpward);`

  Build the connectivity data for entities of type `type`. `rebuild` is an optional argument that tells the method to destroy existing connectivity if present; the default is false. `keepDownward` and `keepUpward` are optional arguments that instruct the mapping to keep downward or upward adjacencies built during the construction of the entities; both are true by default.

- `bool specifyEntity(const EntityTypeEnum type, const intArray &entity);`

  specifiy the entities of a type given by `type` by providing an array of the vertices in each entity. The `entity` array is dimensioned by the number of entities and the maximum number of vertices in that entity type. Each entity in the array is specified by a list of vertex indices terminated by a -1 or by the use of all the available vertices for that entity type.

- `bool buildConnectivity(EntityTypeEnum from, EntityTypeEnum to, bool rebuild);`

  build the connectivity (adjacency) information from entities of type `from` to entities of type `to` . The optional argument `rebuild` forces the mapping to destroy any previous data if set to true; the default is false.

-
  ```
  bool
  specifyConnectivity(const EntityTypeEnum from, const EntityTypeEnum to,
                      const intArray &index, const char *orientation,
                      const intArray &offset);
  ```

  allows the user to specify the adjacencies between `from` and `to` entities. The `index` array contains a list of `to` entities for each `from` entity in compressed form. The beginning of the list of `to` entities for each `from` is specified by the `offset` array.

- `inline bool connectivityExists(EntityTypeEnum from, EntityTypeEnum to) const;`

  returns true if the adjacency information between `from` and `to` has been built already.

- `void deleteConnectivity(EntityTypeEnum from, EntityTypeEnum to);`

  destroys the data associated with the connectivity between `from` and `to` entities.

- `void deleteConnectivity(EntityTypeEnum type);`

  destroys ALL the connectivity information relating to entities of type `type`.

- `void deleteConnectivity();`

  destroys ALL the connectivity information in the mesh.

3 Iterator Classes

There are two iterator classes available : `UnstructuredMappingIterator` , and `UnstructuredMappingAdjacencyIterator` . The former provides iteration through the entities of a particular type and the latter allows the iteration through the entities adjacent to a particular entity. Both skip holes in the entity data structures (unused locations in the arrays) and both optionally skip "ghost" entities by providing an argument to the iterator. Comparisons are also possible between the iterators.

3.1 Common methods and operators

- `void operator++(int);`

  increments the iterator to the next entity.

- `int operator *() const {return e;}`

  returns an unique (amongst the entity type of the iterator) index for the current entity.

- `bool operator==(const UnstructuredMappingIterator & iter) const;`

- `bool`
  `operator==(const UnstructuredMappingAdjacencyIterator & iter) const;`

  return true if two iterators point to the same entity.

- `bool operator!=(const UnstructuredMappingIterator & iter) const`

- `bool`
  `operator!=(const UnstructuredMappingAdjacencyIterator & iter) const`

  return true if two iterators point to different entities.

3.2 `UnstructuredMappingIterator` constructor

The `UnstructuredMappingIterator` is typically initialized by the use of a `UnstructuredMapping` 's `begin` or `end` method. While not intended for common use by the user, the constructor for this class is:

-
  ```
  UnstructuredMappingIterator(const UnstructuredMapping & uns,
                              UnstructuredMapping::EntityTypeEnum entityType_,
                              int position, bool includeGhostEntities );
  ```

The first argument is the `UnstructuredMapping` containing the data to iterate through. `entityType_` designates the the type, or topological dimension, of the entities for iteration. The `position` argument is 0 if an iterator pointing to the beginning of the list is requested and 1 for the end. Finally, if `includeGhostEntities` is true, ghost entities will be skipped during the iteration.

3.3 `UnstructuredMappingAdjacencyIterator` constructor

As with the `UnstructuredMappingIterator` , the `UnstructuredMappingAdjacencyIterator` constructor is intended to be used by methods in the `UnstructuredMapping` class for beginning, ending and querying iterators:

-
  ```
  UnstructuredMappingAdjacencyIterator(const UnstructuredMapping & uns,
                                       UnstructuredMapping::EntityTypeEnum from,
                                       int adjTo,
                                       UnstructuredMapping::EntityTypeEnum to,
                                       int position, bool skipGhostEntities_ = false );
  ```

The first argument is the `UnstructuredMapping` containing the data to iterate through. `from` is an iterator (either a `UnstructuredMappingIterator` or `UnstructuredMappingAdjacencyIterator`) pointing to the entity around which the iteration will take place. The `EntityTypeEnum` variable `to` specifies the desired topological dimension of the adjacent entities. `position` argument is 0 if an iterator pointing to the beginning of the list is requested and 1 for the end. Finally, if `skipGhostEntities` is true, ghost entities will be skipped during the iteration.

3.4 `UnstructuredMappingAdjacencyIterator` orientation

Each entity is given an orientation when it is constructed by the `UnstructuredMapping` . For example, the first vertex index in each edge is always the one with the lowest index value. Each entity maintains information about the orientation of adjacent (upward or downward adjacencies) entities relative to its own definition. For example, a “Face” adjacent to two “Regions” will have its vertices defined counterclockwise when viewed from outside one (the first in the upward adjacency) of the regions. This is by definition the adjacency with a positive orientation. When viewed relative to and from outside the second “Region”, the “Face” will have a clockwise, or negative, orientation. (MAYBE ADD A FIGURE TO DESCRIBE THIS?). In the downward adjacency from the first region to the face the adjacency is positive (the face is oriented correctly relative the the first region) and negative to the second (it is reversed relative to the second). In the upward adjacency, the first “Region” has a positive orientation while the second has negative. The orientation information is provided in the `UnstructuredMappingAdjacencyIterator` by the following method:

- `int orientation() const;`

This method returns 1 for positive orientation and -1 for negative. Use of the adjacency orientation will be demonstrated in the Examples section.

4 The tagging interface

Often it is convenient to tag an entity or group of entities with a piece of data for later use. Common examples would include tagging boundary condition information on boundary entities and adding material property information to groups of entities. A new interface has been created to support these features and consists of the typdefs `entity_tag_iterator` , and `tag_entity_iterator` and the following new methods:

-
  ```
  EntityTag &
  addTag( const EntityTypeEnum entityType, const int entityIndex,
          const std::string tagName,
          const void *tagData, const bool copyTag, const int tagSize );
  ```

  adds a tag onto an entity of type `entityType` at index `entityIndex` . The "name" of the tag, used later for lookups of the tag, is specified by the string `tagName`. An optional argument `tagData` is a void pointer, possibly pointing to user defined data. User defined data can be managed by the tagging system by forcing deep copies of the tagData by setting `copyTag` to true. If `copyTag` is true and `tagData` is a pointer to user allocated data, `tagSize` must specify the size of the tag instance.

-
  ```
  int
  deleteTag( const EntityTypeEnum entityType, const int entityIndex,
             const EntityTag &tagToDelete );
  ```

  deletes the tag associated with an entity of type `entityType` at index `entityIndex` and matching the name of `tagToDelete` .

-
  ```
  int
  deleteTag( const EntityTypeEnum entityType, const int entityIndex,
             const std::string tagToDelete );
  ```

  deletes the tag associated with an entity of type `entityType` at index `entityIndex` and matching the string `tagToDelete` .

-
  ```
  bool
  hasTag( const EntityTypeEnum entityType, const int entityIndex,
  ```

```
        const std::string tag );
```

returns true if an entity has a tag with the name `tag`.

-
  ```
  EntityTag &
  getTag( const EntityTypeEnum entityType,
          const int entityIndex, const std::string tagName);
  ```

  returns a reference to the instance of `EntityTag` associated with a particular entity and tag name.

-
  ```
  void *
  getTagData( const EntityTypeEnum entityType, const int entityIndex,
             const std::string tag );
  ```

  returns a pointer to that data stored by a tag corresponding to a particular name on a given entity defined by `entityType` and `entityIndex` .

-
  ```
  int
  setTag( const EntityTypeEnum entityType, const int entityIndex,
  const EntityTag & newTag );
  ```

  copies the data in `newTag` to the matching tag in the entity defined by `entityType` and `entityIndex` .

-
  ```
  int setTagData( const EntityTypeEnum entityType, const int entityIndex,
                  const std::string tagName,
                  const void *data, const bool copyData=false, const int tagSize=0 );
  ```

  sets the data in a tag on the specified entity. User defined data can be managed by the tagging system by forcing deep copies of the tagData by setting `copyTag` to true. If `copyTag` is true and `tagData` is a pointer to user allocated data, `tagSize` must specify the size of the tag instance.

- `void maintainTagToEntityMap( bool v );`

  if `v` is true, the mapping creates and maintains the list of entities associated with each tag. If false, the mapping destroys this information and does not maintain it until reset to true.

- `bool maintainsTagToEntityMap() const;`

  returns true if the mapping maintains the list of entities with associated with each tag;

- `inline entity_tag_iterator entity_tag_begin(EntityTypeEnum et, int index);`

  returns an iterator to the beginning of the tags associated with an entity defined by `et` and `index`.

- `inline entity_tag_iterator entity_tag_end(EntityTypeEnum et, int index);`

  returns an iterator to the end of the tags associated with an entity defined by `et` and `index`.

- `inline tag_entity_iterator tag_entity_begin(std::string tagName);`

  returns an iterator to the beginning of the entities associated with a tag named `tagName`.

- `inline tag_entity_iterator tag_entity_end(std::string tagName);`

  returns an iterator to the end of the entities associated with a tag named `tagName`.

Note that any of these methods may throw an exception of type `UnstructuredMapping::TagError` . Dereferencing an `entity_tag_iterator` results in a reference to an `EntityTag` . Dereferencing a `tag_entity_iterator` results in a reference to an instance of `UnstructuredMapping::IDTuple` whose straightforward definition is :

```
struct IDTuple
 {
   EntityTypeEnum et; int e;

   inline IDTuple(EntityTypeEnum et_=Invalid, int e_=-1) : et(et_), e(e_) { }
   inline IDTuple( const IDTuple &id ) : et(id.et), e(id.e) { }
   inline ~IDTuple() {}

   inline bool operator< ( const IDTuple & id ) const
   inline bool operator< ( const IDTuple & id )
   inline bool operator== ( const IDTuple & id ) const
   inline bool operator== ( const IDTuple & id )
   inline bool operator!= ( const IDTuple & id ) const
   inline bool operator!= ( const IDTuple & id )
 };
```

`IDTuple's` can be used to provide a shorthand for comparing entities. This class may be merged into `UnstructuredMappingIterator` at some later date.

5 Other useful stuff

Two new public static class members have been added to `UnstructuredMapping` :

```
static aString EntityTypeStrings[];
static aString ElementTypeStrings[];
```

Each of these arrays store string "names" associated with each `EntityTypeEnum` (except `Invalid` whose value is -1) and `ElementType`. These are often useful for diagnostic purposes.

A function to provide integrity checking of meshes and the connectivity built by `UnstructuredMapping` has been created:

- `bool verifyUnstructuredConnectivity( UnstructuredMapping &umap, bool verbose );`

  performs several tests on the connectivity and geometry in the `UnstructuredMapping umap`. If `verbose` is true the function will output any errors as well as some information collected during the integrity checks. The function returns true if there were no errors.

`verifyUnstructuredConnectivity` first checks to make sure the downward and upward adjacencies agree. In other words, it makes sure that each entity is contained in the downward adjacencies of higher dimensional entities and the upward adjacency of lower dimensional ones. These tests ensure that adjacency loops are consistent with one another. The second part of the test is more comprehensive and involves checking the orientations of the adjacencies as well as the geometric validity of the mesh. Geometric validity means that each "Face" in a 2D mesh has a positive area computed with the adjacency information and that each "Region" in 3D has a positive volume. Each "Edge" is also checked for duplicate nodes (zero length edges). During the tests, the volumes or areas are computed several times using the available adjacencies. Inconsistencies in the orientations are detected when these volumes are not the same sign or magnitude. The typical output of this function when the "verbose" option is true looks like:

```
=== VERIFY CONNECTIVITY REPORT ================================
NUMBER OF ERRORS   : 0
NUMBER OF WARNINGS : 0
--- Entity Information ----------------------------------------
Vertex Count : 8121
Edge Count : 34712
Face Count : 47248
Region Count : 20571
--- Element Information ---------------------------------------
triangle Count : 34268
```

```
quadrilateral Count : 12980
tetrahedron Count : 14419
pyramid Count : 2644
triPrism Count : 0
septahedron Count : 0
hexahedron Count : 3508
--- Geometric Information -------------------------------------
Min. Edge : 0.0302402
Max. Edge : 0.245611
Avg. Edge : 0.0889839
Min. Vol : 1.65491e-09
Max. Vol : 0.00461466
Avg. Vol : 0.00017042
===============================================================
```

6 Examples

6.1 Iterating through entities of a given type

Here is an iteration similar to the one in the introduction, except we choose a different entity to iterate through:

```
  UnstructuredMappingIterator edge;
  for ( edge  = umap.begin(UnstructuredMapping::Edge);
        edge != umap.end(UnstructuredMapping::Edge);
        edge++ )
     cout<<"Here is an Edge with index "<<*edge<<endl;
```

6.2 Adjacency iterations

Upward and downward adjacencies are determined by the arguments to the `adjacency_begin/end` functions or the constructor to the `UnstructuredMappingAdjacencyIterator` . The `to` entity type is compared to the type of the `from` iterator for this purpose. Here is a sample downward iteration of the vertices in an edge:

```
  UnstructuredMappingIterator edge = umap.begin(UnstructuredMapping::Edge);
  UnstructuredMappingAdjacencyIterator edgeVert;
  for ( edgeVert  = umap.adjacency_begin(edge, UnstructuredMapping::Vertex);
        edgeVert != umap.adjacency_end(edge, UnstructuredMapping::Vertex);
        edgeVert++ )
     cout<<"Edge "<<*edge<<" has Vertex "<<*edgeVert<<endl;
```

We can define an upwards iteration of the regions, for example, surrounding the edge by altering the second argument of the adjacency begin and end methods:

```
UnstructuredMappingIterator edge = umap.begin(UnstructuredMapping::Edge);
UnstructuredMappingAdjacencyIterator edgeReg;
for ( edgeReg  = umap.adjacency_begin(edge, UnstructuredMapping::Region);
      edgeReg != umap.adjacency_end(edge, UnstructuredMapping::Region);
      edgeReg++ )
   cout<<"Edge "<<*edge<<" has Region "<<*edgeReg<<endl;
```

6.3 Tag all vertices on the boundary of the mesh

This example shows how to use `EntityTypeEnum` 's in a general way to write generic algorithms. This example marks all the vertices that live on the boundary of the mesh with a tag called "boundary vertex".

```
// determine the highest dimensional entity that bounds the mesh
UnstructuredMapping::EntityTypeEnum cellBdyType = umap.getRangeDimension()==2 ?
  UnstructuredMapping::Edge : UnstructuredMapping::Face;

// the next higher entity we will all the ‘‘cell’’
UnstructuredMapping::EntityTypeEnum cellType = ((int)cellBdyType) + 1;

UnstructuredMappingIterator e_iter;
UnstructuredMappingAdjacencyIterator cellIter, vertIter;

// iterate through the bounding entities and determine if they are on the boundary
for ( e_iter=umap.begin(cellBdyType); e_iter!=umap.end(cellBdyType); e_iter++)
  {
    // an e_iter is on the boundary if it only has one neighboring cell
    int nAdj=0;
    for ( cellIter=umap.adjacency_begin(e_iter, cellType);
          cellIter!=umap.adjacency_end(e_iter, cellType); cellIter++ )
       nAdj++;
```

```
        if ( nAdj==1 )
          {
            // we are on a boundary, tag the vertices as such
            for ( vertIter=umap.adjacency_begin(e_iter, UnstructuredMapping::Vertex);
                  vertIter!=umap.adjacency_end(e_iter, UnstructuredMapping::Vertex);
                  vertIter++ )
               if ( !umap.has_tag(UnstructuredMapping::Vertex,
                                 *vertIter, "boundary vertex") )
                 umap.addTag(UnstructuredMapping::Vertex, *vertIter,
                             "boundary vertex", ((void *)*vertIter));
          }

      }
```

The first step determines the highest dimensional bounding entity of the mesh, in 2D this is an Edge, in 3D this is a Face. We then compute the "cell" type, which is the first upward adjacency of the bounding entity (Face in 2D, Region in 3D). We loop through each of the entities of the type that bound the mesh and determine whether they sit on the boundary. If an entity does sit on the boundary, each of its vertices are tagged with a tag named "boundary vertex". The data we store in the tag is just the index of the vertex associated with the tag.

6.4 Computing 2D Face areas

Here is one way to compute the areas of faces in a 2D mesh. Note that this method will also work for arbitrary polygonal faces and could be optimized for tets and quads if necessary:

```
  UnstructuredMapping umap; // get this from somewhere
  UnstructuredMappingIterator edge;
  UnstructuredMappingAdjacencyIterator edgeVert, edgeCell;

  ArraySimple< ArraySimpleFixed<real,3,1,1,1> >
            cellCenters(umap.size(UnstructuredMapping::Face));

  // compute the cell centers with another loop ...

  realArray cellAreas(umap.size(UnstructuredMapping::Face));
  cellAreas = 0;

  // loop through all the edges in the mesh and compute the ``side'' area's
```

```
//  side areas are computed from the edge and the cell centeres.  These areas
//  are then added to the corresponding Face.

for ( edge  = umap.begin(UnstructuredMapping::Edge);
      edge != umap.end(UnstructuredMapping::Edge);
      edge++ )
   {
     ArraySimpleFixed<real,2,1,1,1> edgeVertices[2];

     // in 2D we connect the cell (face) center to
     //    the center of the edge to form the area normals
     ArraySimpleFixed<real,2,1,1,1> edgeVertices[2], edgeCenter;
     int v=0;

     // get the vertices for the edge
     for ( edgeVert  = umap.adjacency_begin(edge,UnstructuredMapping::Vertex);
           edgeVert != umap.adjacency_end(edge,UnstructuredMapping::Vertex);
           edgeVert++ )
       { // we could optimize this loop by getting the edge entities directly
         for ( int a=0; a<rDim; a++ )
            edgeVertices[v][a] = vertices(*edgeVert,a);
       }

     // loop through the adjacent Face's and compute each side area,
     //   adding each area to each Face as we go along
     for ( edgeCell  = umap.adjacency_begin(edge, UnstructuredMapping::Face );
           edgeCell != umap.adjacency_end(edge, UnstructuredMapping::Face );
           edgeCell++ )
       {
         // compute the area of the ``side''
         real area = edgeCell.orientation()*triangleArea2D(edgeVertices[0],
                edgeVertices[1], cellCenters[*edgeCell]);
         // add the area to the cell;
         cellAreas(*edgeCell) += area;
       }
   }
```

6.5 A generic finite volume gradient operator

In this example we construct a generic discrete gradient operator using the finite volume method. We can approximate the gradient of a function u using finite volumes in the usual way with the identity:

$$\int_{\Omega} \nabla u d\Omega = \int_{\partial\Omega} u\mathbf{n}ds \tag{1}$$

Where Ω is some domain bounded by $\partial\Omega$ with an area normal given by $\mathbf{n}ds$. If we replace ∇u by its average , denoted $(\nabla u)_{\Omega}$ we get the following familiar approximation:

$$(\nabla u)_{\Omega} \approx \frac{1}{\Omega} \int_{\partial\Omega} u\mathbf{n}ds \tag{2}$$

There are, of course, a variety of ways to compute this approximation depending on the centering of u on the mesh (Face, Vertex, Edge, etc) and the desired centering of the resulting gradient. If we assume that Ω is represented by polygons in 2D and polyhedra in 3D we can compute the approximation by summing the value of $u\mathbf{n}ds$ on the boundary of a polygon or polyhedra using $\mathbf{n}ds$ computed from the straight line segments (2D) or polygonal surfaces (3D) bounding Ω :

$$(\nabla u)_{\Omega} \approx \frac{1}{\Omega} \sum_{b=1}^{m} u_b(\mathbf{n}ds)_b \tag{3}$$

where the subscript b denotes a facet of the boundary of the polygon or polyhedron. An approximation for u_b is still needed. In some instances, this value may exist at the same centering for u and can be computed by averaging the values on either side of each facet. In other cases, such as a staggered grid, u may exist at a centering with a different adjacency to the gradient (u could exist on the Vertices while ∇u lives on Faces or Regions). In either case, we compute u_b by averaging values adjacent to the facets of Ω :

$$(\nabla u)_{\Omega} \approx \frac{1}{\Omega} \sum_{b=1}^{m} (\frac{1}{n} \sum_{c=1}^{n} u_c^b)(\mathbf{n}ds)_b \tag{4}$$

where u_c^b are the values of u adjacent to facet b.

First we compute the volumes and surface normals required. By noting that each surface is adjacent to two regions, Ω_1 and Ω_2, we can loop through the surfaces computing the value of u_b and adding the result of $u_b(\mathbf{n}ds)_b$ to Ω_1 and subtracting it from Ω_2 (subtract since the area normal relative to Ω_2 points in the opposite direction to Ω_1. The resulting algorithm looks like:

1. Compute the volumes, Ω, of the polygons or polyhedra at the desired centering for $(\nabla u)_{\Omega}$

2. Compute the area normals, $\mathbf{n}ds$, of the facets at the centering corresponding to the boundary of each Ω

3. For each surface `surf`:

   (a) `numberAdjacent` $= 0$

   (b) $u_b = 0$

   (c) For each u, (denoted by `u_cell`), adjacent to `surf`

      i. $u_b += u_{\texttt{u\_cell}}^{\texttt{surf}}$

      ii. `numberAdjacent` $++$

   (d) $u_b /=$ `numberAdjacent`

   (e) $u_{\Omega_1} += \frac{1}{\Omega_1} u_b (\mathbf{n}ds)_{\texttt{surf}}$

   (f) $u_{\Omega_2} -= \frac{1}{\Omega_2} u_b (\mathbf{n}ds)_{\texttt{surf}}$

The code for this algorithm using the new `UnstructuredMapping` iterator interface is given in the function `computeGradient`. `centering` refers to the centering of the function u; `surfaceCentering` the centering of the surface normals; and `gradientCentering` refers to the centering of `gradu` .

```
void computeGradient(UnstructuredMapping &umap,
                     UnstructuredMapping::EntityTypeEnum centering,
                     UnstructuredMapping::EntityTypeEnum surfaceCentering,
                     UnstructuredMapping::EntityTypeEnum gradientCentering,
                     realArray &u, realArray &surfaceNormals, realArray &volumes,
                     realArray &gradu)
{

  UnstructuredMappingIterator surf;
  UnstructuredMappingAdjacencyIterator u_cell, g_cell;

  gradu.redim(umap.size(centering), umap.getRangeDimension());
  gradu = 0;

  for ( surf  = umap.begin(surfaceCentering);
        surf != umap.end(surfaceCentering);
        surf++ )
    {
      int nC=0;
      real uOnSurf=0;
      for ( u_cell  = umap.adjacency_begin(surf, centering);
            u_cell != umap.adjacency_end(surf, centering);
            u_cell++ )
        {
          uOnSurf += u(*u_cell);
          nC++;
        }

      uOnSurf /= real(nC);

      for ( g_cell  = umap.adjacency_begin(surf, gradientCentering);
            g_cell != umap.adjacency_end(surf, gradientCentering);
            g_cell++ )
        for ( int a=0; a<umap.getRangeDimension(); a++ )
          gradu(*g_cell,a) +=
              g_cell.orientation()*surfaceNormals(*surf,a)*uOnSurf/volumes(*g_cell);
    }
}
```

This function can be used to compute Vertex centered gradients using Vertex centered values for u by calling:

```
computeGradient(umap,
                UnstructuredMapping::Vertex, UnstructuredMapping::Edge,
                UnstructuredMapping::Vertex,
                vertexCenteredScalar, edgeAreaNormals, vertexVolumes,
                vertexCenteredGradient);
```

It can also be used to compute “cell” (Face or Region) centered gradients using:

```
computeGradient(umap,
                cellType, UnstructuredMapping::EntityTypeEnum( ((int)cellType)-1 ),
                cellType,
                cellCenteredScalar, cellSurfNormals, cellVolumes,
                cellCenteredGradient);
```

Yet another example places u at the vertices but computes $(\nabla u)_\Omega$ at Face or Region (Cell) centers:

```
computeGradient(umap,
                UnstructuredMapping::Vertex, UnstructuredMapping::EntityTypeEnum( ((int)cellT
                cellType,
                vertexCenteredScalar, cellSurfNormals, cellVolumes,
                cellCenteredGradientFromNodeU);
```